《Rust语言与内存安全设计》

第12讲 智能指针

课程负责人：陈文清 助理教授
chenwq95@mail.sysu.edu.cn

2023年11月15日

上节课IO项目回顾

1. 接收命令行参数
2. 读取文件
3. 重构（模块化和错误处理）
4. 测试驱动开发（Test-Driven Development）
5. 结合环境变量 （请自行查看官网文档）
6. 将错误消息写入标准错误而不是标准输出

本节课内容

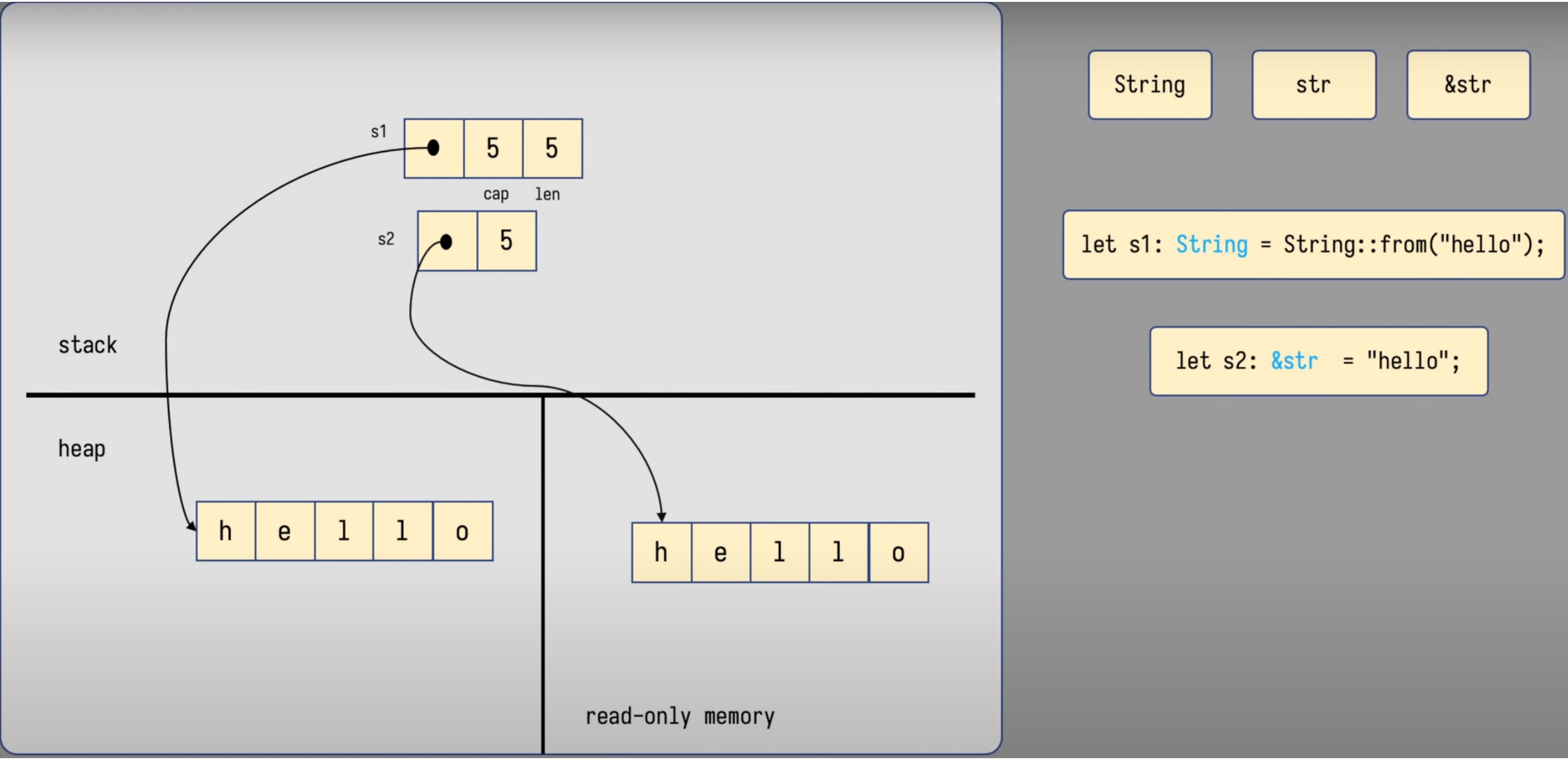
智能指针

Rust中的指针类型

- 引用——安全的指针
- 原始指针
- 智能指针
  - Box<T>
  - Rc<T>
  - Arc<T>
  - Cell<T>
  - RefCell<T>

Rust中的指针类型

- **引用（Reference）——安全的指针**
  - 类似C语言中的指针，通过“&”或“&mut”符号；



➢ **String和 &str（字符串切片）**

Rust中的指针类型

■ 原始指针（Raw Pointers)

- ➢ 与引用一样，原始指针可以是不可变的或可变的，分别写为 *const T 和 *mut T*；
- ➢ *星号(*) 不是解引用运算符*；它是类型名的一部分
- ➢ Unsafe Rust

- 允许同时拥有mutable和immutable借用
- 并不能保证能指向valid memory
- 允许空指针
- 没有实现自动的内存清理；
- (关于unsafe rust之后会详细展开)

```
fn main() {
    let mut num = 5;
    let r1 = &num as *const i32;
    let r2 = &mut num as *mut i32;
    unsafe{
        *r2 = *r2 + 1;
    }
    println!("{:?}", r1);
    println!("{:?}", r2);
    println!("{:?}", num);
}
```

Rust中的指针类型

■ 智能指针（Smart Pointers）

- 智能指针是一种类似于指针的数据结构，同时具有附加的元数据和功能。
- 智能指针通常使用结构体实现。与普通结构体不同，智能指针实现了 Deref 和 Drop 特性。

Rust中的指针类型

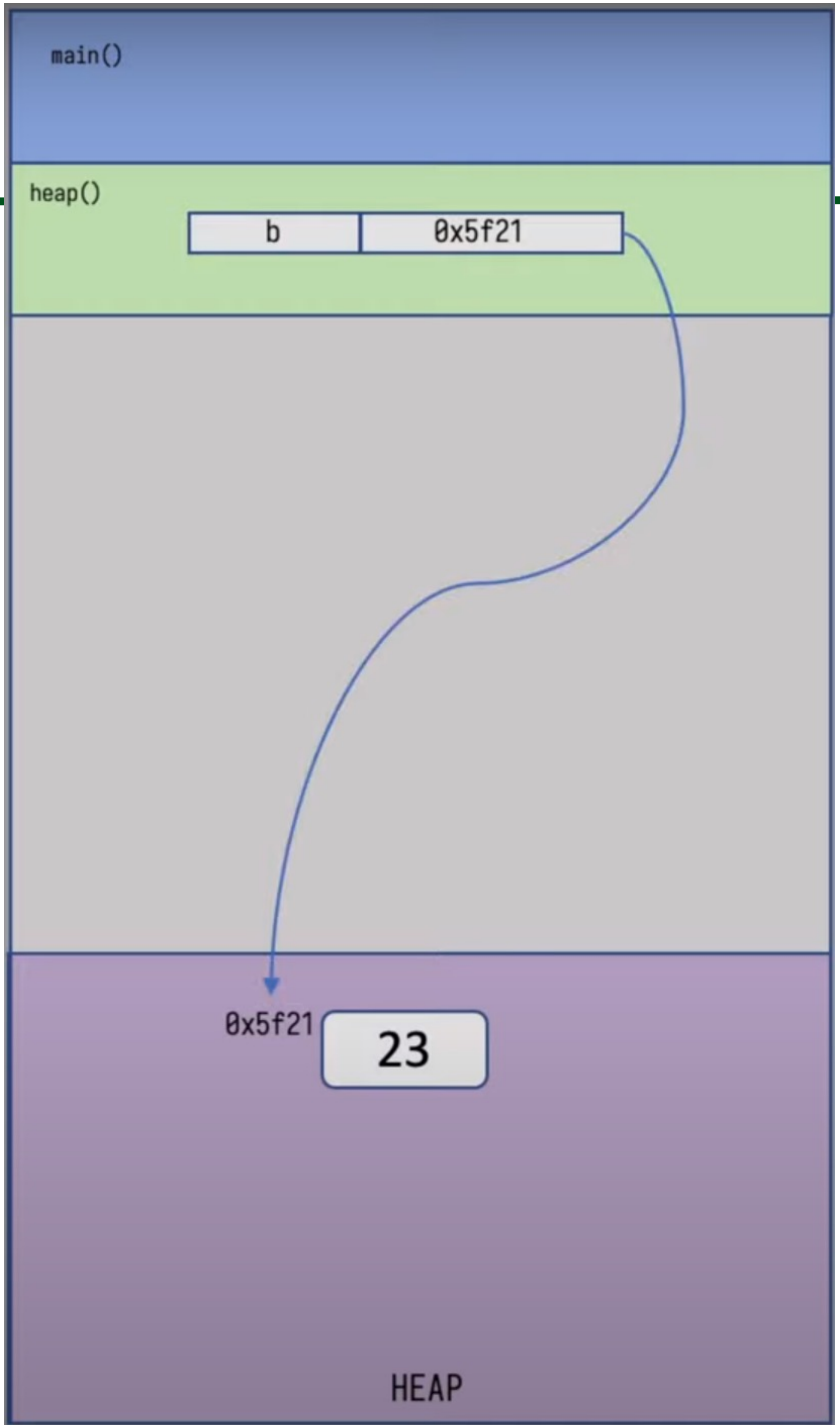
■ 智能指针（Smart Pointers)

- Drop Trait允许你自定义在智能指针实例超出作用域时运行的代码。（执行 drop 方法）
- Deref Trait允许智能指针结构体的实例表现得像一个引用，这样你可以编写代码以处理引用或智能指针。（解引用）

Rust中的指针类型

■ 智能指针（Smart Pointers）

- Box<T>： Boxes allow you to store data on the heap rather than the stack.
- 例：创建一个递归链表




```rust
fn main() {
    let result = heap();
}

fn heap() -> Box<i32> {
    let b = Box::new(23);
    b
}
```

- 回顾关于heap和stack上为Box<T>分配内存的案例

快速回顾：什么是链表？

讨论：

- 你会如何用C或C++实现一个链表类？
  - 你需要什么结构？
  - 你会提供什么样的方法？
  - 你的测试代码看起来会是什么样子？
  - 我们一直在谈论的内存错误，可能会出现什么问题？
- 根据您目前对 Rust 的了解，您认为在 Rust 中实现链表会有什么挑战？

快速回顾：什么是链表？

C++ :

```
struct Node {
    int value;
    Node* next;
}
```

```
int main() {
   Node* first = (Node*)malloc(sizeof(Node));
   first->value = 1;
   Node* second = (Node*)malloc(sizeof(Node));
   second->value = 2;
   first->next = second;

   /* do stuff (e.g., print the list) */

   free(first);
   free(second);
}
```

在Rust 中定义Node……?

C++ :

```
struct Node {
    int value;
    Node* next;
}
```

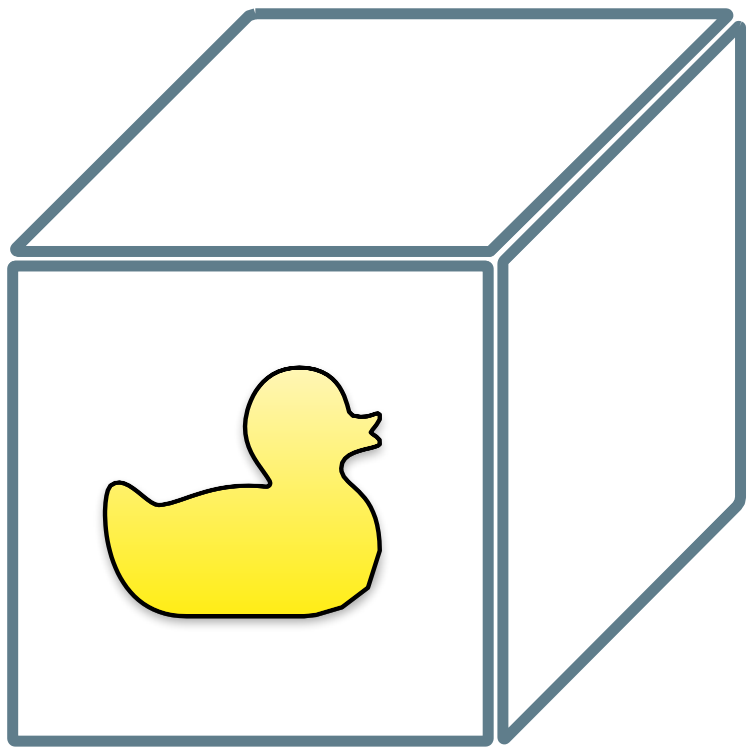
```
struct Node {
    value: i32,
    next: Node, /* won't work! recursive def. */
}
```

```
struct Node {
    value: i32,
    next: &Node, // not what we want! `&` does not create a pointer.
                 // – it implements "borrowing", which doesn't
                 // really apply here.
}
```

```
struct Node {
    value: i32,
    next: /* pointer to a node…? */
}
```

Box in Rust

- 创建一个Box
- Box放在堆上
- 任何东西都可以放在Box里
- Box 拥有盒子里面值的所有权。当 Box 超出范围 -> Box里面的值被销毁。
- 相同的事情像C++里面的*unique_ptr 指针*：
  - "一个智能指针通过指针拥有和管理另一个对象，并在 unique_ptr 超出作用域时处理该对象。"



https://doc.rust-lang.org/rust-by-example/std/box.html

Box in Rust

```
struct  Node {
    value：i32，
}
```

Type：Box<Node>

```
fn main（）｛
    let node = Box::new(Node {value: 1});
    println!("{}" , node.value);
}
```

在堆上声明和分配节点

- 变量“node”拥有 Box<Node>
- 当“节点”不再使用时，Box 会（自动）被销毁
  - 编译器插入对 Box 的 drop 函数的调用。
- 当Box被销毁时，Node对象也被销毁

在 Rust 中定义Node：我们需要什么？

```
struct Node { value: i32,
      next: Box<Node>,
}
```

使用节点：一个元素链接列表

```
struct Node {
    value: i32, next:
     Box<Node>,
}

fn main() {
    let node = Box::new( Node {
                    value: 1,
                    next:   /* equiv. of nullptr…?*/,
                    });
}
```

使用Options

```
struct Node {
    value: i32,
    next: Option<Box<Node>>,
}
```

Could be Some or None
If Some, will contain Box<Node>

```
fn main() {
    let node = Box::new( Node {
                            value: 1,
                            next: None
                        });
}
```

Last element in list?
`next` is None

This has made its way back into C++ — see [std::optional]

让我们列一个更长的list ~~~ take 1

does not compile

```
struct Node {
   value: i32,
   next: Option<Box<Node>>,
}


fn main() {
    let mut first = Box::new( Node { value: 1, next: None });
    let second = Box::new( Node { value: 2, next: None });
    first.next = second;
}
```

Reminder: we want to change `first`, so explicitly make it mutable

让我们列一个更长的list ~~~ take 1

This SHOULD be an Option…

but you’re giving me a Box?????

```
|       first.next = second;
|                    ^^^^^^
|                    |
|                    expected enum `std::option::Option`, found struct `std::boxed::Box`
|                    help: try using a variant of the expected enum: `Some(second)`
|
= note: expected enum `std::option::Option<std::boxed::Box<_>>`
           found struct `std::boxed::Box<_>`

or[E0308]: mismatched types
```

让我们列一个更长的list ~~~ take 1

Compiles!

```
struct Node {
    value: i32,
    next: Option<Box<Node>>,
}


fn main() {
    let mut first = Box::new( Node { value: 1, next: None });
    let second = Box::new( Node { value: 2, next: None });
    first.next = Some(second);   // This is now Option<Box<Node>>
}
```

让我们列一个更长的list ~~~ take 1

does not compile

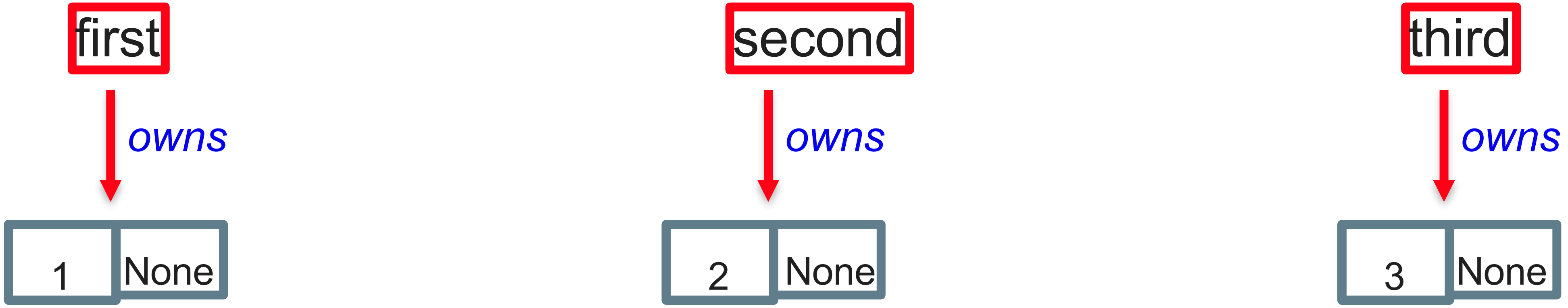
```
struct Node {
   value: i32,
   next: Option<Box<Node>>,
}


fn main() {
    let mut first = Box::new( Node { value: 1, next: None });
    let mut second = Box::new( Node { value: 2, next: None });
    let third = Box::new( Node { value: 3, next: None });
    first.next = Some(second);
    second.next = Some(third);
}
```

让我们列一个更长的list ~~~ take 1

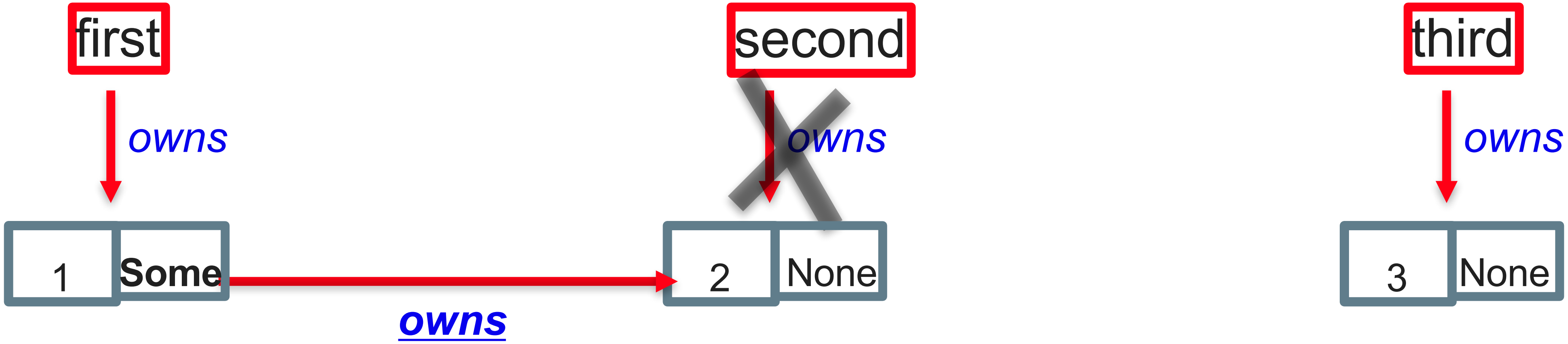
```
error[E0382]: assign to part of moved value: `*second`
  --> src/main.rs:12:5
   |
8  |     let mut second = Box::new(Node {value: 2, next: None});
   |         ---------- move occurs because `second` has type `std::boxed::Box<Node>`, whic
h does not implement the `Copy` trait
...
11 |     first.next = Some(second);
   |                       ------ value moved here
12 |     second.next = Some(third);
   |     ^^^^^^^^^^^ value partially assigned here after move
   |
```

这里发生了什么事？



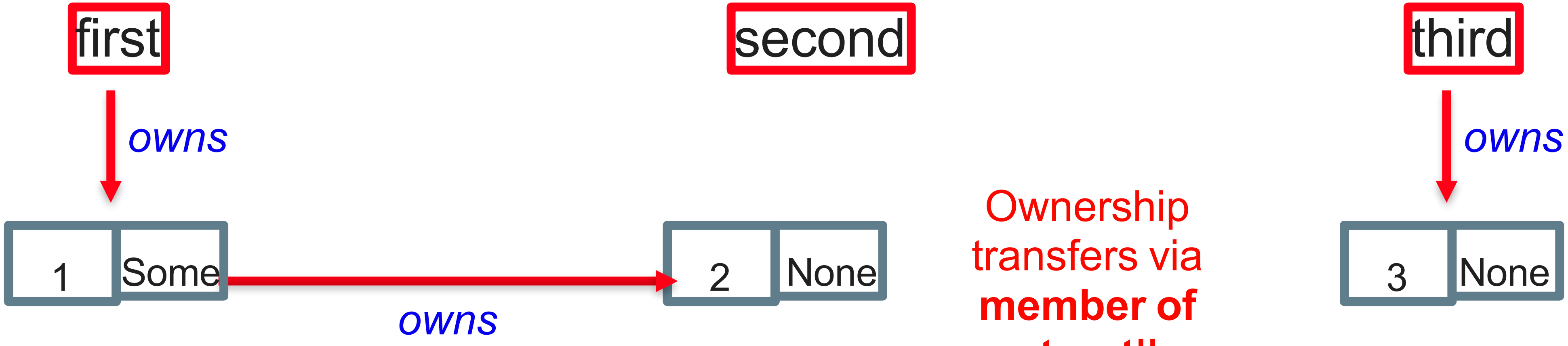

```
fn main() {
    let mut first = Box::new( Node { value: 1, next: None });
    let mut second = Box::new( Node { value: 2, next: None });
    let third = Box::new( Node { value: 3, next: None });
    first.next = Some(second);
    second.next = Some(third);
}
```

这里发生了什么事？




```
fn main() {
    let mut first = Box::new( Node { value: 1, next: None });
    let mut second = Box::new( Node { value: 2, next: None });
    let third = Box::new( Node { value: 3, next: None });
    first.next = Some(second);
    second.next = Some(third);
}
```

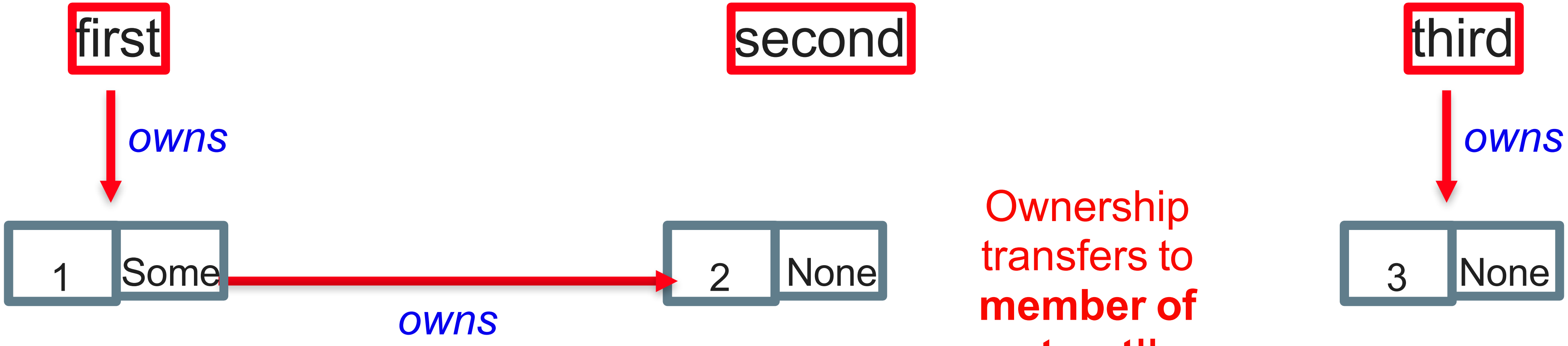
这里发生了什么事？



Ownership transfers via **member of struct!!**

```
fn main() {
    let mut first = Box::new( Node { value: 1, next: None });
    let mut second = Box::new( Node { value: 2, next: None });
    let third = Box::new( Node { value: 3, next: None });
    first.next = Some(second);
    second.next = Some(third);
}
```

这里发生了什么事？

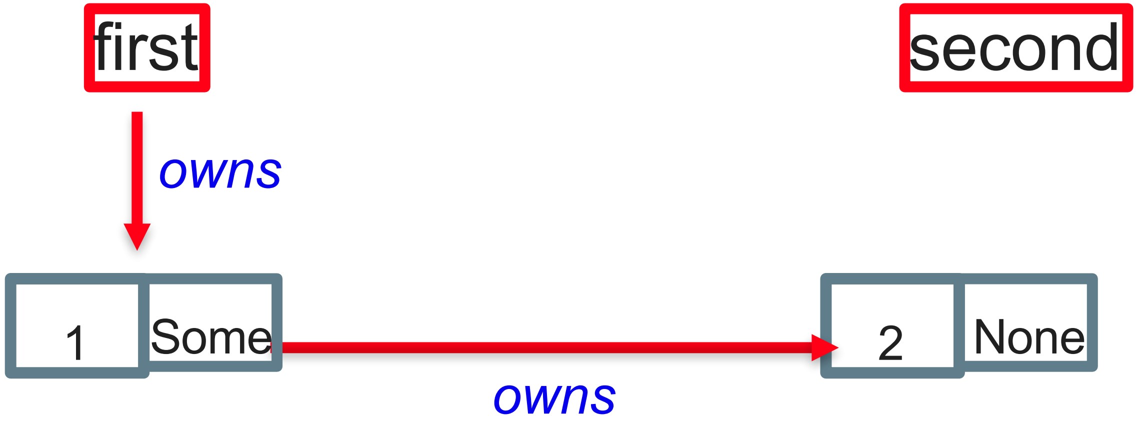


Ownership transfers to **member of struct**!!

```
fn main() {
    let mut first = Box::new( Node { value: 1, next: None });
    let mut second = Box::new( Node { value: 2, next: None });
    let third = Box::new( Node { value: 3, next: None });
    first.next = Some(second);
    second.next = Some(third);
}
```

Error — can no longer access Box<Node> via variable `second`

“Chain of ownership（所有权链）”



- Implication: when `first` is dropped (destroyed):
  - First node of list is dropped,
  - …so Option (in Node struct) is dropped,
  - …so Box (in Option) is dropped,
  - …so second Node (in Box) is dropped.
- Everything is cleaned up :)

- …But we can’t use `second` anymore to access this node.
- *These are the type of issues that can get really annoying in Rust :(*

让我们列一个更长的list ~~~ take 1

compiles

```
struct Node {
    value: i32,
    next: Option<Box<Node>>,
}


fn main() {
    let mut first = Box::new( Node { value: 1, next: None });
    let mut second = Box::new( Node { value: 2, next: None });
    let third = Box::new( Node { value: 3, next: None });
    second.next = Some(third); // swap order of these lines
    first.next = Some(second); // use `second` to access node
                               // before it loses ownership.
}
```

Let’s print the list!

C++ :

```
struct Node {
    int value;
    Node* next;
}
```

C++:

```
int main() {
    Node* first = (Node*)malloc(sizeof(Node));
    first->value = 1;
    Node* second = (Node*)malloc(sizeof(Node));
    second->value = 2;
    first->next = second;

    Node *curr = first;
    while (curr != NULL) {
        printf("%d\n", curr->value);
        curr = curr->next;
    }

    free(first);
    free(second);
}
```

goal: this, but in Rust…

我们如何将其转换为 Rust?

- Rust 中没有 “ 指针” ；`curr` 应为哪种类型？
- 我们可能希望 `curr` 引用。以 “ first” 节点开始，但我们不希望 “ first”失去节点的所有权。（我们不希望一旦 “ curr” 不再被使用，列表就会被释放！）
- 我们的循环条件是什么？（我们怎么知道什么时候已经到达终点？）

C++:

```
Node *curr = first;
while (curr != NULL) {
    printf("%d\n", curr->value);
    curr = curr->next;
}
```

Let’s print the list!

```
fn main() {
    let mut first = Box::new( Node { value: 1, next: None });
    let mut second = Box::new( Node { value: 2, next: None });
    let third = Box::new( Node { value: 3, next: None });
    second.next = Some(third);
    first.next = Some(second);

    let mut curr = /* something */;

}
```

make `curr` mutable, because we’re going to reassign it

Let’s print the list!

```
fn main() {
    let mut first = Box::new( Node { value: 1, next: None });
    let mut second = Box::new( Node { value: 2, next: None });
    let third = Box::new( Node { value: 3, next: None });
    second.next = Some(third);
    first.next = Some(second);

    let mut curr = Some(&first);

}
```

`curr` is an **Option<&Box<Node>>**

- **Option**: can be `Some` or None
    - Use `None` to indicate end of List
- **&Box<Node>:**
    - If Some: <&Box<Node>>
    - Want to take the Box by reference (why might this be important?)
    - Box “contains” heap-allocated Node

Let’s print the list!

```
fn main() {
    let mut first = Box::new( Node { value: 1, next: None });
    let mut second = Box::new( Node { value: 2, next: None });
    let third = Box::new( Node { value: 3, next: None });
    second.next = Some(third);
    first.next = Some(second);

    let mut curr: Option<&Box<Node>> = Some(&first);

}
```

提醒：如果你愿意的话，你可以显式地为变量写入类型。否则，Rust 编译器会为我们进行类型推断。

Let’s print the list!

```
fn main() {
    let mut first = Box::new( Node { value: 1, next: None });
    let mut second = Box::new( Node { value: 2, next: None });
    let third = Box::new( Node { value: 3, next: None });
    second.next = Some(third);
    first.next = Some(second);

    let mut curr = Some(&first);
    while curr.is_some() {
      // print value

      // update curr
       }
  }
```

Option 提供了 is_some() 和 is_none() 方法。我们希望在 curr 有某个值的情况下继续循环。（与 C++ 示例中的 while curr != NULL 相同的逻辑。）

Let’s print the list!

```
fn main() {
    let mut first = Box::new( Node { value: 1, next: None });
    let mut second = Box::new( Node { value: 2, next: None });
    let third = Box::new( Node { value: 3, next: None });
    second.next = Some(third);
    first.next = Some(second);

    let mut curr = Some(&first);
    while curr.is_some() {
        println!("{}", curr.value);
        // update curr
    } }
}
```

does not compile

`curr` is an Option — `.value` isn’t valid.

Let’s print the list!

```
fn main() {
    let mut first = Box::new( Node { value: 1, next: None });
    let mut second = Box::new( Node { value: 2, next: None });
    let third = Box::new( Node { value: 3, next: None });
    second.next = Some(third);
    first.next = Some(second);

    let mut curr = Some(&first);
    while curr.is_some() {
        println!("{}", curr.unwrap().value);
        // update curr
    }
}
```

compiles!

- 如果 curr 是 Some，则提取其值。
- 否则，引发 panic（使程序崩溃）。
- 在这里，可以相对安全地假设 curr 是 Some，因为我们刚刚在前一行进行了检查。

提醒/回顾：Option、Enum、Unwrap

```
println!("{}", curr.unwrap().value);
```

Std Rust lib:

```
enum Option {
    Some(<T>),
    None,
}
```

Stores a value

- Option 可以是 Some 或 None。
- 如果是 Some，它会存储一个对象（这里是 &Box<Node>）。
- curr.unwrap() 的意思是：
    - 如果 curr 是 Some，则返回 Some 中的内容。
    - 如果 curr 是 None，则引发 panic。

Let's print the list!

```
fn main() {
    let mut first = Box::new( Node { value: 1, next: None });
    let mut second = Box::new( Node { value: 2, next: None });
    let third = Box::new( Node { value: 3, next: None });
    second.next = Some(third);
    first.next = Some(second);

    let mut curr = Some(&first);
    while curr.is_some() {
        println!("{}", curr.unwrap().value);
        curr = curr.unwrap().next;
    }
}
```

does not compile

```
struct Node {
    value: i32,
    next: Option<Box<Node>>,
}
```

- curr.unwrap() 给我们一个节点
- Node.next 给了我们 Option<Box<Node>>，为什么这不是我们想要的？

Introducing `as_ref()`

- Converts &Option<T> into Option<&T>
- If provided Option is None, returns None
- E.g.:

```
let mut curr = Some(&first);
while curr.is_some() {
    println!(“{}”, curr.unwrap().value);
    curr = (&curr.unwrap().next).as_ref();
}
```

- curr.unwrap().next 给我们的是 Option<Box<Node>>。
- 应用 & 给我们的是 (&Option<Box<Node>>)。
- 应用 as_ref() 给我们的是 Option<&Box<Node>>。
- 如果 curr.unwrap().next 是 None，as_ref() 返回 None。

Let's print the list!

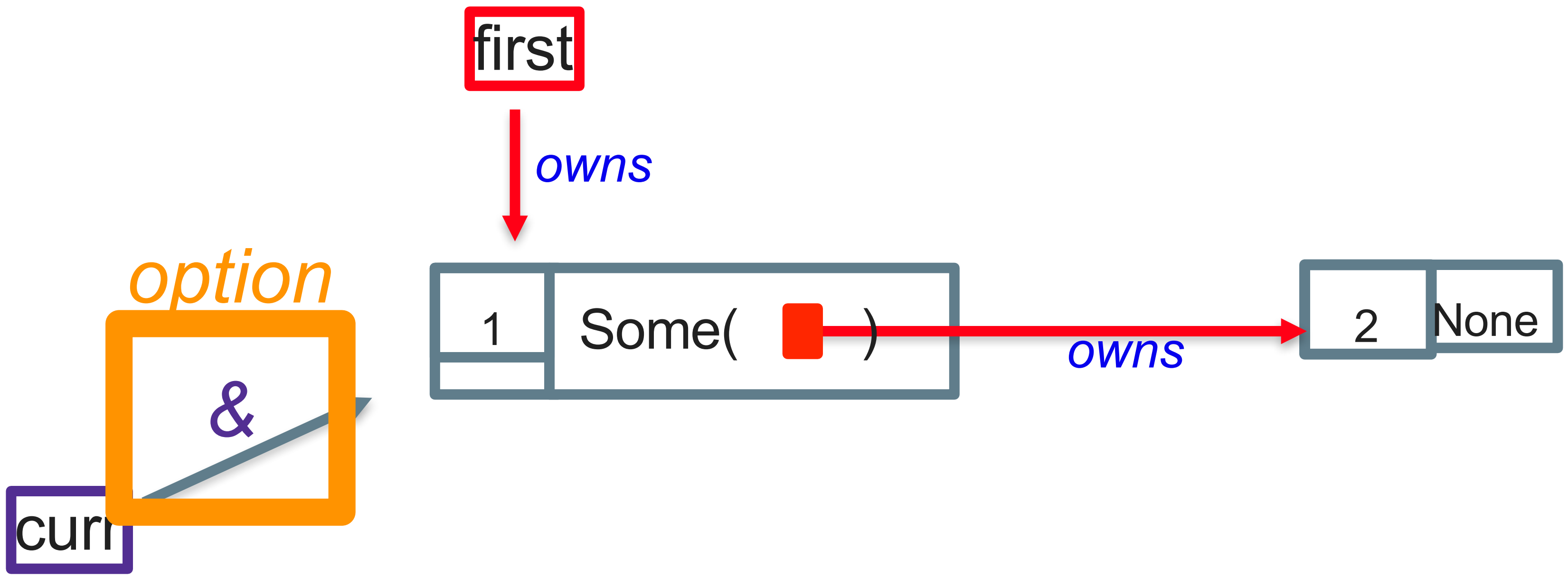
```
fn main() {
    let mut first = Box::new( Node { value: 1, next: None });
    let mut second = Box::new( Node { value: 2, next: None });
    let third = Box::new( Node { value: 3, next: None });
    second.next = Some(third);
    first.next = Some(second);

    let mut curr = Some(&first);
    while curr.is_some() {
        println!("{}", curr.unwrap().value);
        curr = (&curr.unwrap().next).as_ref();
    }
}
```

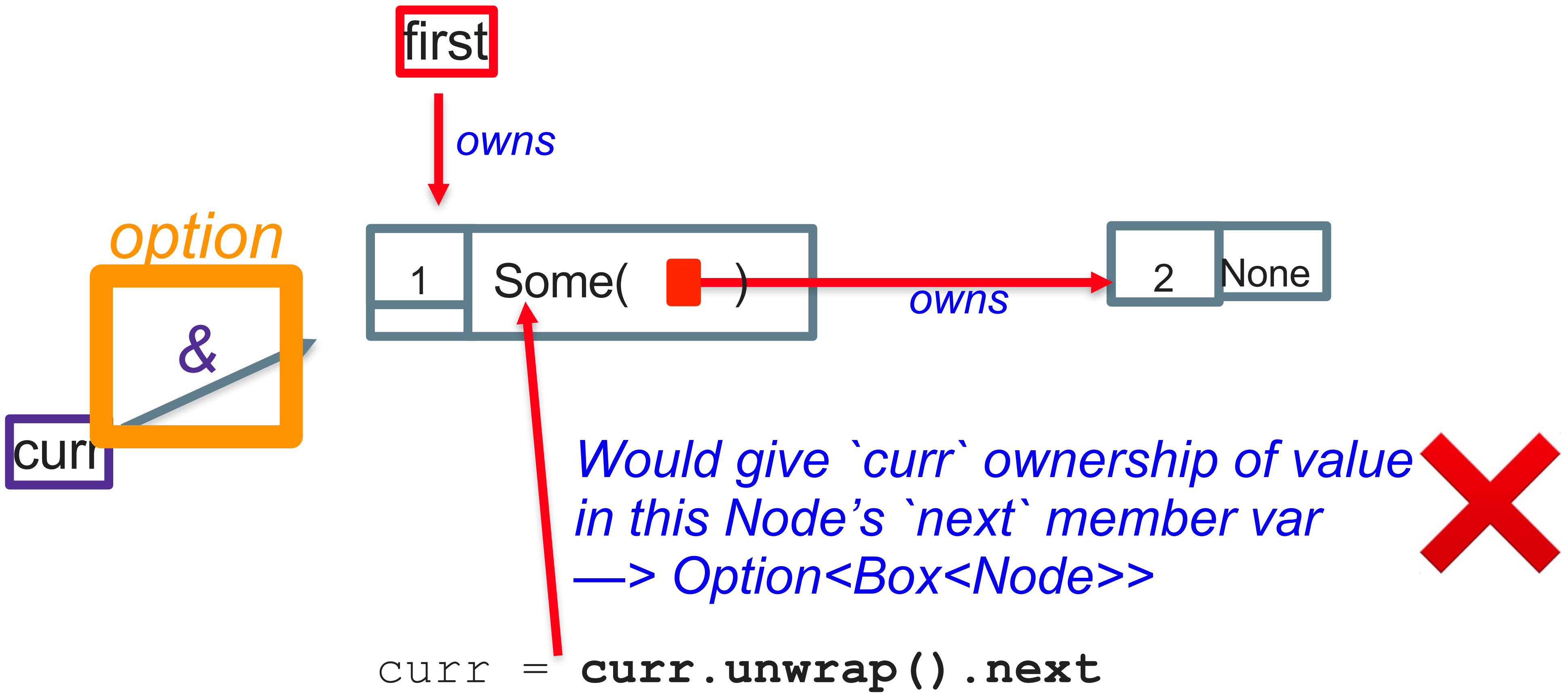
compiles

Option<&Box<Node>>

Changing `curr`, illustrated

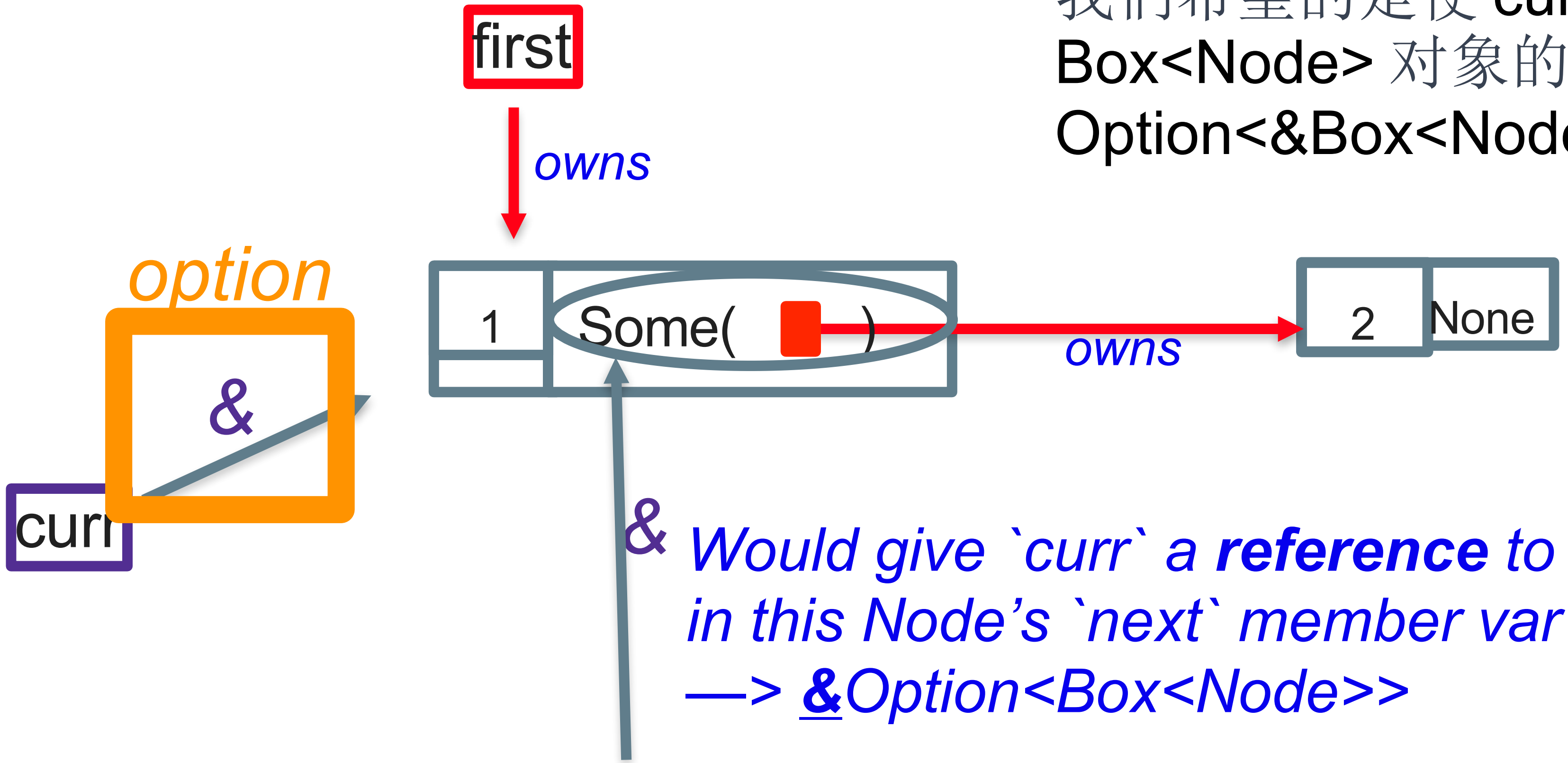


Changing `curr`, illustrated



as_ref()：一个[有点不太好的]示例

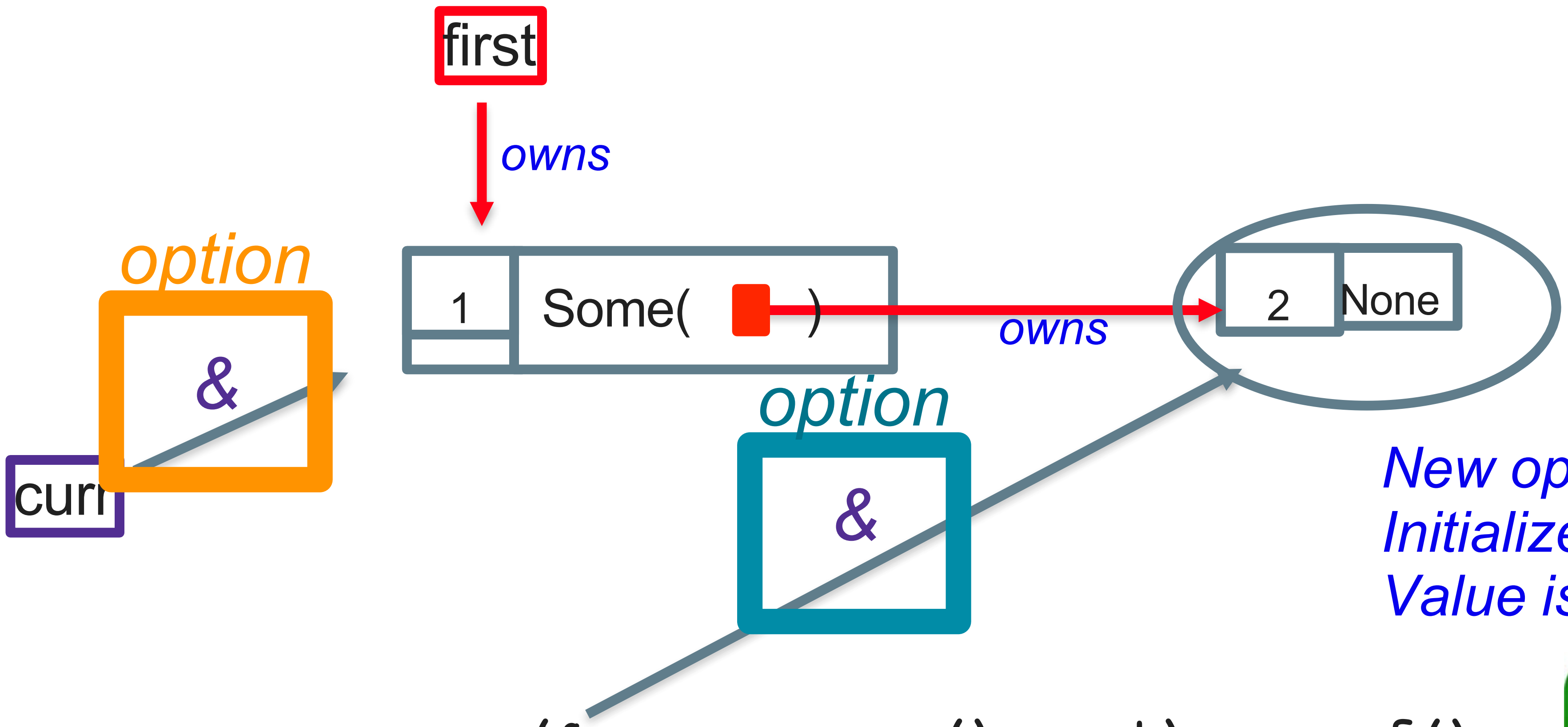
我们希望的是使 curr 变为对这个 Box<Node> 对象的引用的Option —> Option<&Box<Node>>



*Would give `curr` a **reference** to the value in this Node's `next` member var —> <u>**&**</u>Option<Box<Node>>*

```
curr = &curr.unwrap().next
```

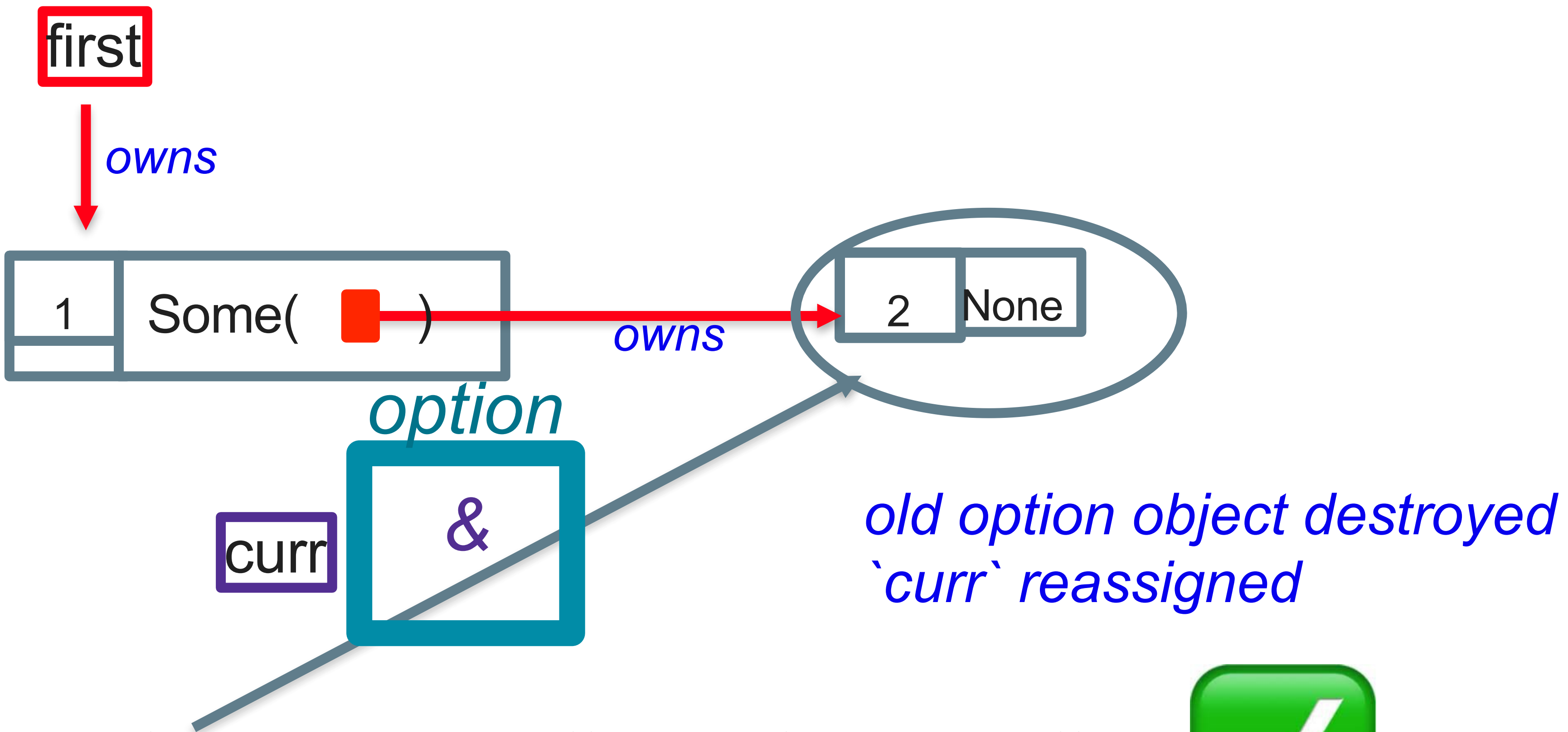
as_ref()：一个[有点不太好的]示例



New option object is created
Initialized as Some
Value is &Box<Node>

```
curr = (&curr.unwrap().next).as_ref()
```

as_ref()：一个[有点不太好的]示例




```
curr = (&curr.unwrap().next).as_ref()
```

✅

Rust中的指针类型

■ 智能指针（Smart Pointers)

- Deref Trait允许智能指针结构体的实例表现得像一个引用，这样你可以编写代码以处理引用或智能指针。（解引用）
- Drop Trait允许你自定义在智能指针实例超出作用域时运行的代码。（执行 drop 方法）

Rust中的指针类型

■ 智能指针（Smart Pointers)

➢ The Deref trait allows an instance of the smart pointer struct to behave like a reference （解引用）

```
fn main() {
    let x = 5;
    let y = &x;

    assert_eq!(5, x);
    assert_eq!(5, *y);
}
```

```
fn main() {
    let x = 5;
    let y = Box::new(x);

    assert_eq!(5, x);
    assert_eq!(5, *y);
}
```

智能指针实现了`std::ops::Deref`特征，解引用会默认调用`deref`方法
请测试`y.deref()`和`*(y.deref())`

https://doc.rust-lang.org/std/ops/trait.Deref.html

自定义智能指针

```rust
struct MyBox<T>(T);

impl<T> MyBox<T> {
    fn new(x: T) -> MyBox<T> {
        MyBox(x)
    }
}
```

```rust
fn main() {
    let x = 5;
    let y = MyBox::new(x);

    assert_eq!(5, x);
    assert_eq!(5, *y);
}
```

自定义智能指针

```rust
struct MyBox<T>(T);

impl<T> MyBox<T> {
    fn new(x: T) -> MyBox<T> {
        MyBox(x)
    }
}
```

```rust
fn main() {
    let x = 5;
    let y = MyBox::new(x);

    assert_eq!(5, x);
    assert_eq!(5, *y);
}
```

得到的编译错误是:

```
error[E0614]: type `MyBox<{integer}>` cannot be dereferenced
  --> src/main.rs:14:19
   |
14 |     assert_eq!(5, *y);
   |                   ^^
```

自定义智能指针

```rust
struct MyBox<T>(T);

impl<T> MyBox<T> {
    fn new(x: T) -> MyBox<T> {
        MyBox(x)
    }
}
```

```rust
use std::ops::Deref;


impl<T> Deref for MyBox<T> {
    type Target = T;

    fn deref(&self) -> &T {
        &self.0
    }
}
```

当我们在示例 15-9 中输入 `*y` 时，Rust 事实上在底层运行了如下代码：

```rust
*(y.deref())
```

函数和方法的隐式解引用强制转换

解引用强制转换（*deref coercions*）是 Rust 在函数或方法传参上的一种便利。解引用强制转换只能工作在实现了 Deref trait 的类型上。解引用强制转换将一种类型（A）隐式转换为另外一种类型（B）的引用，因为 A 类型实现了 Deref trait，并且其关联类型是 B 类型。比如，解引用强制转换可以将 &String 转换为 &str，因为类型 String 实现了 Deref trait 并且其关联类型是 str。代码如下：

```rust
#[stable(feature = "rust1", since = "1.0.0")]
impl ops::Deref for String {
    type Target = str;

    #[inline]
    fn deref(&self) -> &str {
        unsafe { str::from_utf8_unchecked(&self.vec) }
    }
}
```

Rust中的指针类型

■ 智能指针（Smart Pointers)

- ➢ Drop Trait允许你自定义在智能指针实例超出作用域时运行的代码。（执行 drop 方法）

```rust
struct CustomSmartPointer {
    data: String,
}

impl Drop for CustomSmartPointer {
    fn drop(&mut self) {
        println!("Dropping CustomSmartPointer with data `{}`!", self.data);
    }
}
```

```rust
fn main() {
    let c = CustomSmartPointer {
        data: String::from("my stuff"),
    };
    let d = CustomSmartPointer {
        data: String::from("other stuff"),
    };
    println!("CustomSmartPointers created.");
}
```

介绍智能指针之前，让你为某个结构体实现Drop Trait;
注意：观察对象Drop的顺序。

Box<T>

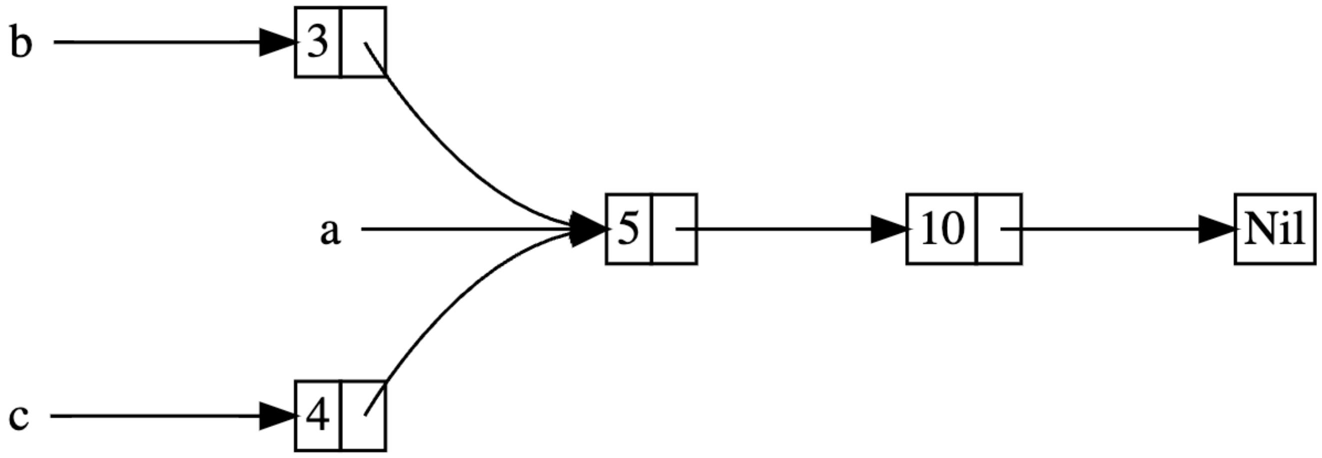
- 具有指向堆内存块的唯一指针
- Box<T>有哪些局限性？

Rc<T>

- 如果我想拥有指向同一堆内存块的多个指针，该怎么办？
- 回想一下借用规则：可以有多个不可变引用，或者最多有一个可变引用。
- Rc <T>允许你对堆内存块拥有多个不可变引用（即我们不能修改这块内存）
    - 我们为什么需要这个？
    - 答：Rust 的借用检查规则！
- 注意：如果你创建了引用循环，可能会导致内存泄漏！（如果你需要引用循环，你需要将其他智能指针类型加入到组合中）

Example: Adding Multiple Views to Our List

- 如果我们希望我们的链表能够相互“相交”，以便它们可以在数据结构不可变的情况下共享某些部分，该怎么办？（这是函数数据结构中常见的范式）
- 这可以让我们看到数据结构的“历史”！
- 这些有时被称为持久的数据结构；
- Playground示例
  - 开始
  - 结束



图片：https://doc.rust-lang.org/book/ch15-04-rc.html

Rust中的指针类型

■ 智能指针（Smart Pointers)

- Rc<T>： Reference Counted Smart Pointer（引用计数智能指针，用于单线程）。
- Arc<T>： Atomic Reference Counting Smart Pointer（原子引用计数智能指针，用于多线程，在之后多线程课程中介绍）。

Rust中的指针类型

■ 智能指针（Smart Pointers）

- Rc<T>： Reference Counted Smart Pointer（引用计数智能指针，用于单线程）。
  - 通过使用 Rust 类型 Rc<T> 来明确启用多所有权；
  - 想象 Rc<T> 就像家庭房间里的一台电视。
  - 当有人进来看电视时，他们打开它。
  - 其他人可以进入房间并观看电视。
  - 当最后一个人离开房间时，他们关闭电视，因为它不再被使用。

Rust中的指针类型

■ 智能指针（Smart Pointers)

- Rc<T>： Reference Counted Smart Pointer（引用计数智能指针，用于单线程）。
  - 通过使用 Rust 类型 Rc<T> 来显式启用多所有权；
  - 把 Rc<T> 想象成家庭房间里的一台电视。
  - 当有人进来看电视时，他们打开它。（创建）
  - 其他人可以进入房间并观看电视。（引用+1）
  - 当最后一个人离开房间时，他们关闭电视，因为它不再被使用。（引用次数减为0时，释放）

Rust中的指针类型

■ 智能指针（Smart Pointers）

➢ Rc<T>

```
enum List {
    Cons(i32, Box<List>),
    Nil,
}

use crate::List::{Cons, Nil};

fn main() {
    let a = Cons(5, Box::new(Cons(10,
Box::new(Nil))));
    let b = Cons(3, Box::new(a));
    let c = Cons(4, Box::new(a));
}
```

```
enum List {
    Cons(i32, Rc<List>),
    Nil,
}

use std::rc::Rc;
use crate::List::{Cons, Nil};

fn main() {
    let a = Rc::new(Cons(5,
Rc::new(Cons(10, Rc::new(Nil)))));
    let b = Cons(3, Rc::clone(&a));
    let c = Cons(4, Rc::clone(&a));
}
```

Rust中的指针类型

■ 智能指针（Smart Pointers)

➢ Rc<T>

```
fn main() {
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
    println!("count after creating a = {}", Rc::strong_count(&a));
    let b = Cons(3, Rc::clone(&a));
    println!("count after creating b = {}", Rc::strong_count(&a));
    {
        let c = Cons(4, Rc::clone(&a));
        println!("count after creating c = {}", Rc::strong_count(&a));
    }
    println!("count after c goes out of scope = {}", Rc::strong_count(&a));
}
```

```
count after creating a = 1
count after creating b = 2
count after creating c = 3
count after c goes out of scope = 2
```

注意：上述引用为 immutable references.

help: trait `DerefMut` is required to modify through a dereference, but it is not implemented for `Rc<T>`

Q & A

Thanks!